**Post-Office-Queue** Summary Report

Methodology: I decided to implement a linked list which operates as a FIFO queue. This I believed would be a better representation of how such a system would ordinarily work in reality.

The reason for using Linked lists over arrays is that I do not have to manage the storage location of the actual structures, I only have to keep track of the pointers to these structs. Moving these between different structure such as moving customers between the queue and the service points is far less memory intensive. The hope was that in future implementations this would allow me to run more advanced simulations or even run simpler ones more quickly.

Structures: I created customer structures which act as nodes in the LL. These store several attributes among which are a unique identifier which was useful during troubleshooting, the waiting tolerance limit here on out referred to as a customer’s patience. This also contains a counter for the time elapsed since the customer entered the Post Office before being served. There is also a pointer to the next node.

To simplify enQueue and deQueue methods as well as counting the number of nodes in the queue I have created a separate structure called queue which stores two pointers, one to the front (head) of the LL and the other for the rear (tail) of the queue. This makes referencing the LL much easier.

Next, I have a servicePoint structure which stores the SP unique id as well as the time till the current customer has finished being served and a pointer to the customer. I felt it would be more efficient to use pointers in this implementation so that I do not have to copy the entire customer into the service point structure, moreover the pointer will be of fixed size so no need to reallocate memory each time.

Randomness: I experimented with four different distributions, namely Uniform, Normal (Gaussian), Poisson, Gamma.

The Poisson distribution will give positive discrete integer whereas all the others will need to be cast to integer values. Gamma is a left skewed distribution so could be used to randomly generate values which favour the lower end of the distribution. Uniform has an equal probability of each value being selected so could be compared to Normal distribution bell curve to see what effect if any results.

We should consider that it would be best to keep either the number of service points constant between simulations so that we can determine the effect of randomising customer’s patience limits and the length of time they spent being served or so that we can determine the optimum number of service points required to serve a fixed number of customers. This also applies to the random elements of the program where the limits and parameters of those functions should remain fixed when attempting to validate efficiency of the overall system. On separate investigations these variables can be altered to look at how each factor affects the computation. It would be wise to use a large number of runs to ensure that the simulation averages are representative of the system.

Project Structure As mentioned above my project relies on three main structures, namely customers, queue, and service points. As part of my design decision, I decided to create separate modules to handle all the functionality associated with the queue and its customers as well as the service points. For the queue this meant having distinct methods for enQueuing and dequeuing customers, along with functionality for counting the length to ensure the maxQueue limit had not been exceeded.

As part of this process, I deliberated over whether the newNode function belonged in a separate customer module but decided against this on the ground that any ordinary queue module would be expected to contain its own methods for creating newNodes and it would be far easier for anyone reviewing this code to have the customer structure bundled with the queue function. The same was true for the function designed to remove customers from the queue when they have reached their waiting tolerance limit and for this, I would argue it is simple an extension of the deQueue function.

The main simQ.c module is the simplest module containing only the functionality for transferring the input parameters from the readInputFile modules to the simulations module while relaying the parameters to the writeOutputToFile module.

From here the simulations module calls the various functions from multiple different modules required to run the simulations. This also involves displaying the iterative data for each time unit when running a single simulation or alternatively tallying the totals across multiple simulations before averaging the results and sending these to writeOutputToFile to be saved in the output file.

File handling is managed both by readInputFile which checks for the correct number of parameters and validate the input file. These values are updated in simQ using pointers. The alternative would have been to create a separate parameters struct and return this from the readInputFile module. This seemed unnecessarily complex considering we at most require six parameters to run the sim.

In order to store our results in the output file we make use of the writeOutputToFile module. This is relatively slow as each time the write function is called we want to ensure that the opened file is correctly closed to prevent any data corruption as a result of running out of memory at runtime. Furthermore, this module is responsible for ensuring that the user cannot accidentally overwrite any files by checking that the given output file name does not match any existing documents. There are also three type casting function to ensure that the data passed to this module is correctly converted to an array of characters without resulting in float conversion or unsigned integer errors. The file is opened each time in append mode which will create the file in the first instance and tack on any additional lines as the program continues through the simulations.

Another key module is the servicePoints, here we combine the structure definition as well and the methods for counting the number of SPs in use at the current time interval. Moreover, we have functionality checking when each customer has finished being served and can leave the post office. This then allows us to transfer a new customer from the queue to the free service point. Both the simulation and queue modules are responsible for incrementing the tallies found within the simulations module so that we get accurate results.

The final module is responsible for the random element of the simulation. The choose distribution function can be passed a mean value as well as a distribution identifier to generate a random number based on this information. This module is also responsible building the GSL library in the first instance and make use of an existsGSL integer which ensure the generator is created only once. This greatly increases the efficiency of the program as less memory is occupied by GSL and makes running each simulation faster as there is no time wasted in regenerating and tearing down the structure each iteration. Rather, the GSL pointer is initialised within the simulation modules and passed as a double pointer so that its value can be updated within the randomDistributions module.

Within the respective header files I have ensure all the necessary function prototypes and structure relating to each module are defined and imported into the various element of the program with as little overlap as possible. Finally, I have ensured that, before any variables go out of scope, all memory allocations made while running the simulation are freed so that the memory can be reused.

Distributions I ran my simulation with multiple distributions and found that the Poisson distribution was more suitable in a large number of cases. My program has the ability to select Uniform, Normal/Gaussian, Poisson and Gamma distributions for each of the three variables averageNewCustomerPerInterval, averageTimeTakenToServeCustomer as well as averageWaitingToleranceOfCustomer. The desired mean value can be passed in from the inputFile and the standard deviation uses a constant value of three. I believed this S.D. to be a good compromise between generating enough randomness to get interesting results while still approximating the generate numbers close to the mean. There would be scope in the future to also have the S.D. value passed in from the inputFile.

For my testing I ran each of the distributions 100 times to generate a list of candidate numbers. I found that the Gamma distribution while still giving the correct mean value, had some vast outliers likely due to the skew of the curve that underlies the distribution. Uniform distribution would have provided some interesting results though were less useful when it came to doing the investigation due to the equal probability of each value with the given range being selected.

The Normal distribution was another contender though it did require me to check for negative values and regenerate values when this occurred. This I thought might slow down the simulation on larger runs and so was reserved for smaller values.

I decided upon using Poisson for the wider investigation due to its relative stability. The values never deviated too drastically from the mean and so I believed would provide a good representation for the overall results. The script for these can be found in *extras/experimental/allDistributions.c*

Testing and Investigation See *multipleSimsOutput.txt* or *singleSimOut.txt* for output files where numSims > 1 and numSims = 1.

Number of Simulations: 1000  
  
Max Queue Length: 20  
  
Number of Service Points: 3  
  
Time until close: 100  
  
Average waiting tolerance limit: 5  
  
Average time taken to serve customer: 5  
  
Average new customers per interval: 3  
  
Average number of fulfilled customers: 42.259998  
Average number of unfulfilled customers: 366.95001  
Average number of timed-out customers: 69.153999  
Average wait for fulfilled customers: 8.365949  
Average time after close until customers leave: 17.252001

Through my testing I have found that provided the maxQueueLength is reasonably large and the number of customers is relatively small, if the number of service points is approximately 25 or greater, then the average amount of time spent waiting by each fulfilled customer does not vary drastically. This is likely because once the queue is full provided the new number of customers joining each interval is similar or greater than the number of service points and accounts for the number of customers likely to leave the queue as timed-out customers then the length of the queue will remain relatively fixed and so the overall number of customers served in a single day is roughly the same. We would however of course see a vastly increased number of timed-out customers.

From these results we can infer that for a business such as a Post Office we can there is little added benefit with having vast numbers of service points, and in reality, the queue length would only be as big as the floor space of the building allows. This might allow an employer to determine the optimum number of service points Therefore they can employ and hence pay the minimum number of people that will allow them to do the greatest amount of business relative to their spending on staff wages. This optimisation of resources and personnel could lead to significant improvement all while potentially reducing the customers waiting time to a more reasonable figure.

This overlooks online services where customers are likely to have a greater waiting tolerance limits due to not physically spending their time in one place but rather being able to simply reopen a tab when it is there turn.

Following on from the above increasing the number of service points has the greatest effect on reducing the number of seconds after closing time that all customers leave the Post Office. We can reasonably expect that provided the number of units till closing time is sufficiently large to allow the queue to become fully saturated, then with a greater number of service points we are able to more quickly clear the remainder of the queue near the end of the simulation.

The biggest factor in the ration of fulfilled to unfulfilled customers is the length of the queue compared to the new arrivals per interval. With a longer queue we are likely to see a greater number of timed out customers because customers are not turned away from joining the queue. This results in the problem being transferred elsewhere, specifically an increase in time-out customers as there will be more customers failing to reach the front of the queue before giving up and leaving.

In determining the average number of customers that can be served in one day as previously discussed the number of customers arriving per interval was found to be a less significant factors than the number of service points due to the limitation on queue size. Leading on from this I decided to experiment with varying the average time take to serve any given customer. As expected, this was the most important factor in determining the total daily customers served because for any given number of units till closing time the number of customers leaving fulfilled would be greatest when the time taken at the service point was the smallest. Of course, this still relies on a reasonable number of customers arriving in each interval such that the queue remaining partially fully. This has the added benefit of reducing customer waiting time which does not account for the time spent being served but only for the time spent waiting in the queue before accessing the service points.